CMPSC-265
Data Structures and Algorithms

Zaihan Yang
zyang13@suffolk.edu

Department of Math and Computer Science
Suffolk University

Fall 2019

Notice

- HW9 will be posted today, and will be due on next Tuesday midnight (11.59pm)
- Will have you take-home Quiz 3
  - due on Sunday 11.59pm
  - Submit onto Blackboard

Recap

- Binary Search Tree
- Basic operations:
  - Search for a node
  - Insert a node
  - Find the node with the minimum value in the BST
  - Find the node with the maximum value in the BST
  - Delete a node (consider three cases)
  - Traversal of the BST (the same as general tree):
    - Level-order
    - Pre-Order
    - In-Order
    - Post-Order

Learning Topics

- Binary Heap

Binary Heap

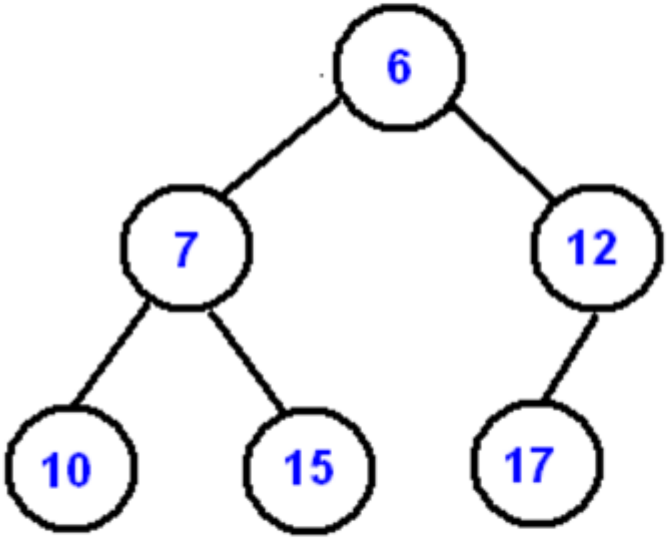
- Binary heap is a complete binary tree with heap ordering property.
  - Complete binary tree
  - Convenient be represented in an array
  - The depth is (int)$\log_2 N$
  - Heap ordering property
  - Suitable to implement the priority queue:
    - Both removal and insertion are in O(logN), and is easy to implement.

Heap Ordering Property

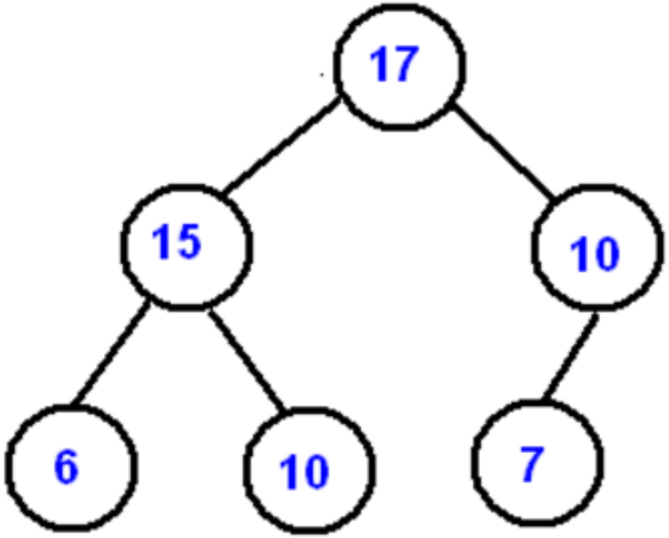
- Two types:
  - Max-heap ordering property:
    - Every node’s key is greater than (or equal to) the keys of its children:
    - Therefore the maximum value is at the root.
  - Min-heap ordering property:
    - Every node’s key is smaller than (or equal to) the keys of its children
    - Therefore the maximum value is at the root.

Heap Ordering Property

Min-Heap



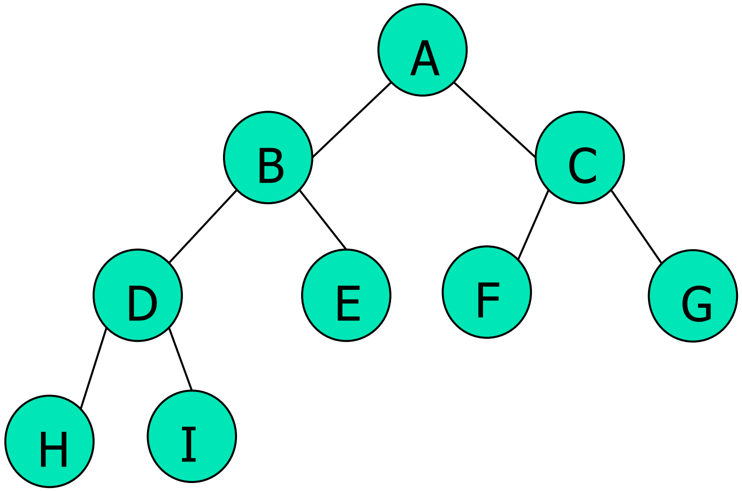
Max-Heap

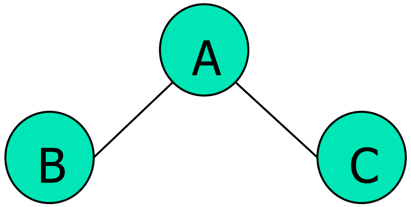


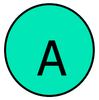
- Either the maximum or the minimum value is at the root, and that's why it gets its name as "heap"
- It is not completely sorted structure, only partially ordered. There is no particular relationship among the nodes on any given level, even among the siblings.

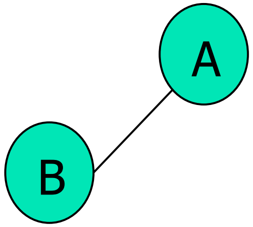
Complete Binary Tree

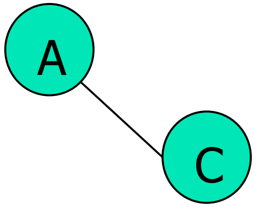
- If each level of ***t*** is full with the possible exception of the deepest one k. If level k is not full, all leaves must be filled from the left to the right.







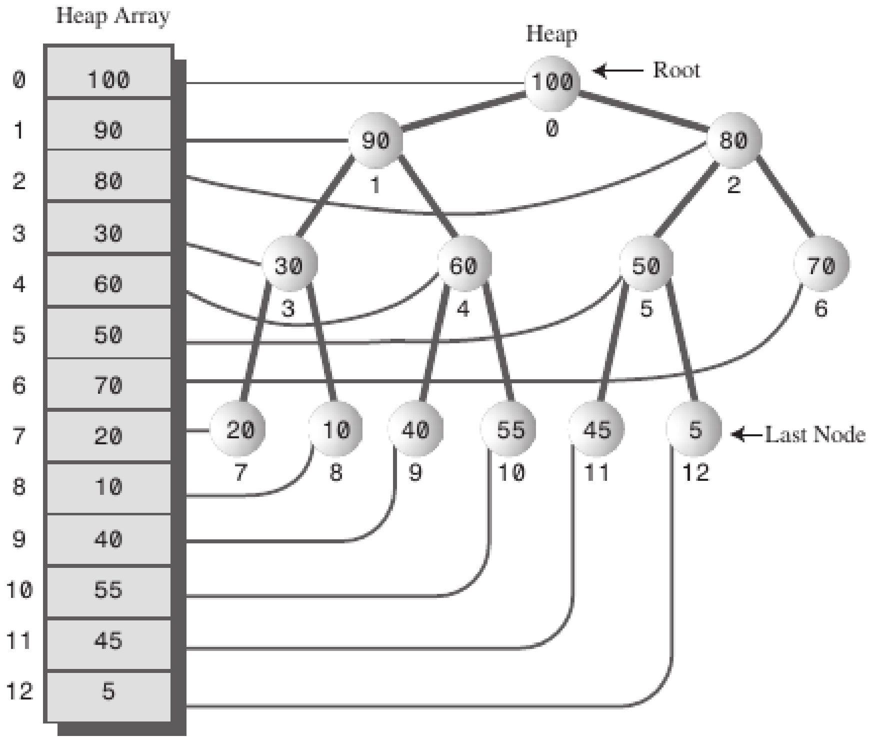




NOT

Array-based Representation of Heap

- Heap is a *complete* binary tree, So, using an array does not waste space.



How to access children of the node at index *i*?

- Left child: 2*i +1
- Right child: 2*i +2

Parent of *i* : (i -1)/2

Robert Lafore. 2002. Data Structures and Algorithms in Java (2 ed.). Sams, Indianapolis, IN, USA. Page 581

Heap Class

```
class Heap
{
    private Node[] heapArray;
    private int maxSize; // size of array
    private int currentSize; // number of nodes in array
    // --------------------------------------------------
    public Heap(int mx) // constructor
    {
        maxSize = mx;
        currentSize = 0;
        heapArray = new Node[maxSize]; // create array
    }
    // ---------------------------------------------------
    public boolean isEmpty()
    { return currentSize==0; }
    // ---------------------------------------------------
    public boolean insert(int key)
    {…}
    // ---------------------------------------------------
    public Node remove() // delete item with max key
    {…}
}
```

```
class Node
{
   private int iData;
   // --------------------------------
   public Node(int key) //constructor
   { iData = key; }
   // --------------------------------
   public int getKey()
   { return iData; }
   // --------------------------------
   public void setKey(int id)
   { iData = id; }
   // --------------------------------
} // end class Node
```

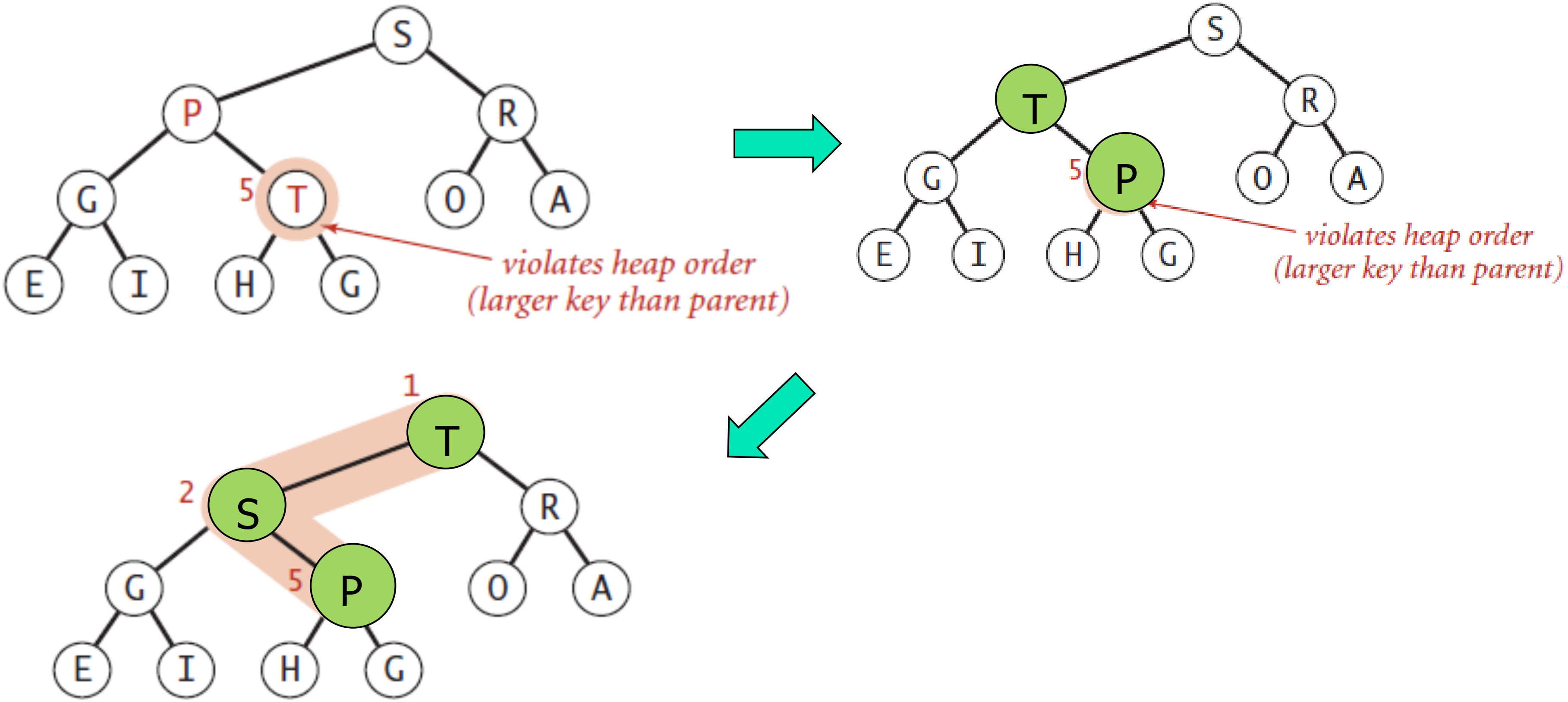
Robert Lafore. 2002. Data Structures and Algorithms in Java (2 ed.). Sams, Indianapolis, IN, USA. Page 592

Heap operations

- Sometimes the heap property is violated, and we have to travel through the heap, modifying the heap as required so that to ensure the heap property is satisfied everywhere.
- Reheapifying (restoring heap order)
    - Two operations:
        - Bottom-up reheapify (swim / trickle up / percolation up)
        - Top-down reheapify (sink / trickle down / percolation down)

Bottom-up reheapify (swim/ trickleUp)

When a node’s key is larger than its parent’s key (A<B<C…..<Z)



violates heap order (larger key than parent)

violates heap order (larger key than parent)

Bottom-up reheapify (swim / trickleUp)

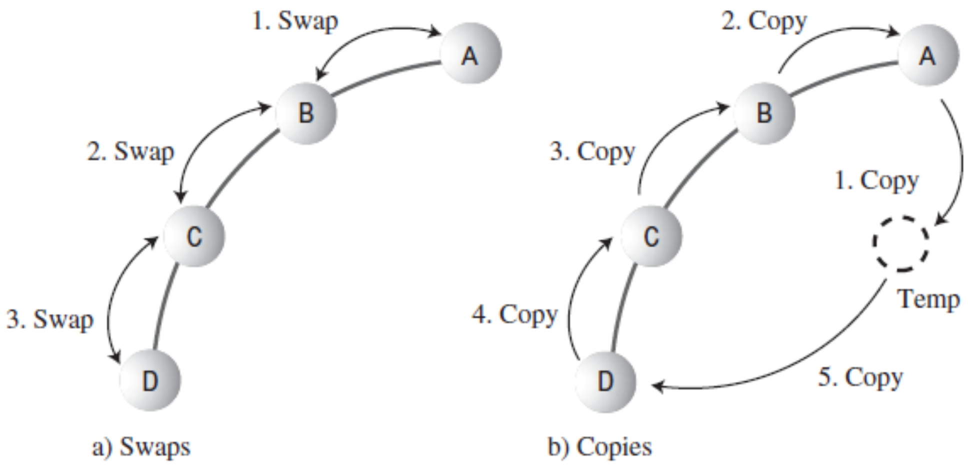
```
private void trickleUp(int i) {
   // reached the root
   if (i==0) return;

   int parent=(i-1)/2;
   if (heapArray[i]>heapArray[parent])
   {
      swap(i,parent); //swap heapArray[i] and heapArray[parent]
      trickleUp(parent); //continue with parent
   }
}
```

What is the time complexity?

O(log n)

Minor Improvement-Copy Instead of Swap

- Every swap takes 3 copies
- We can improve the trickleUp() method by using copies instead of swaps
- In the following example: 9 copies Vs 5 copies. If the tree is tall, there is more improvement.



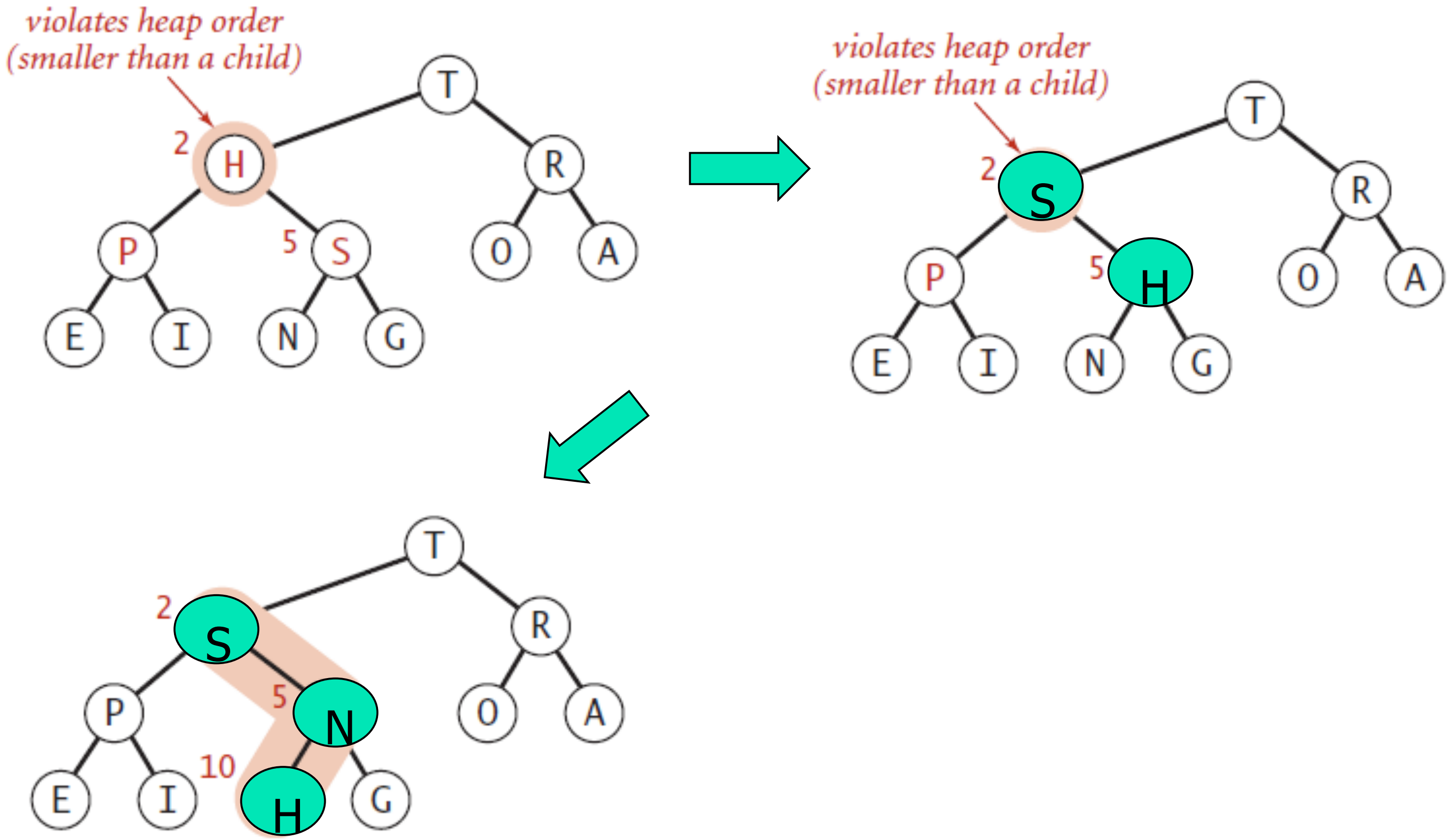
Robert Lafore. 2002. Data Structures and Algorithms in Java (2 ed.). Sams, Indianapolis, IN, USA. Page 586

Minor Improvement-Copy Instead of Swap

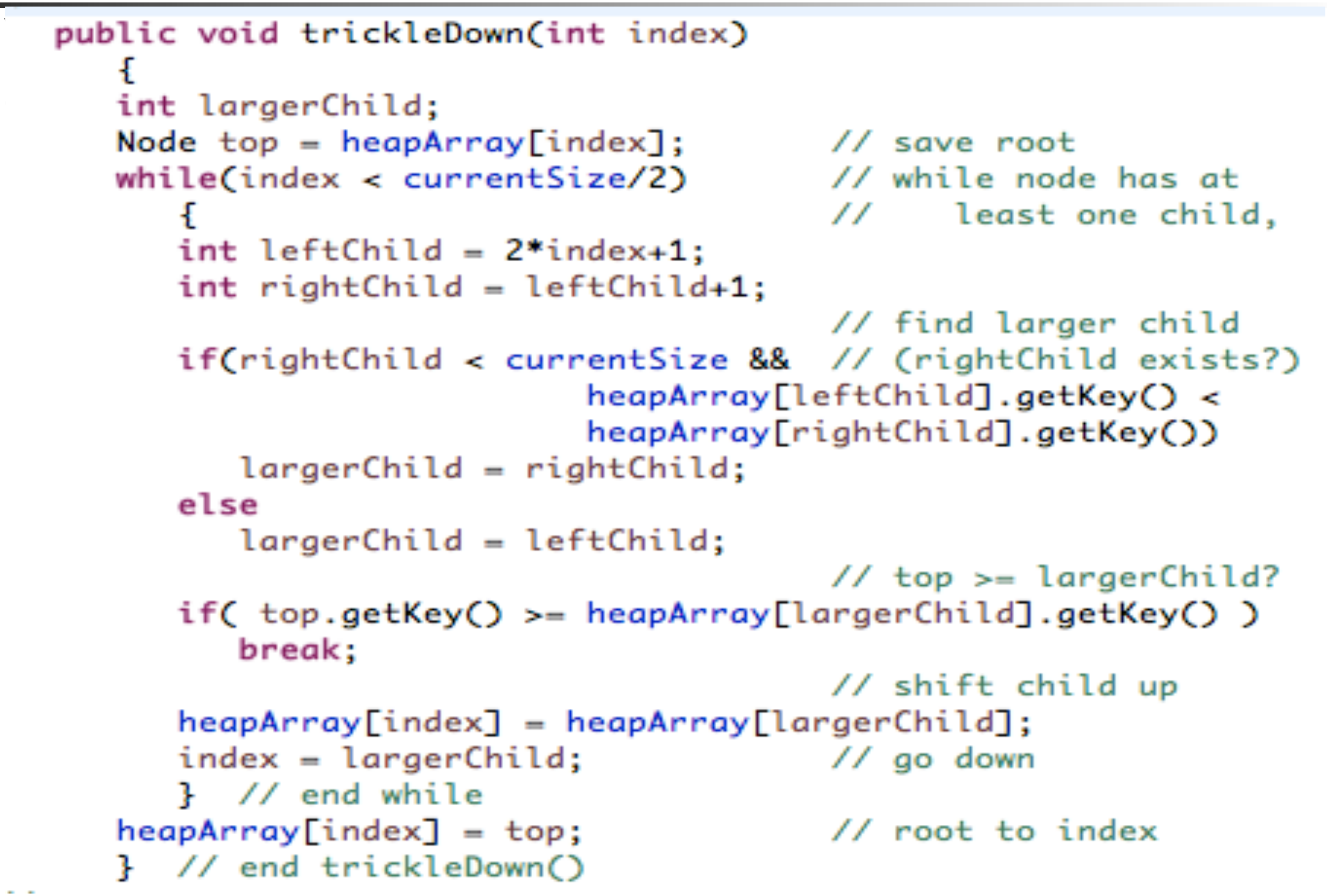
```
private void trickleUp(int index)
{
   int parent = (index-1) / 2;
   Node bottom = heapArray[index];

   while( index > 0 && heapArray[parent].getKey() <
bottom.getKey() )
   {
     heapArray[index] = heapArray[parent];
     index = parent;
     parent = (parent-1) / 2;
   }

   heapArray[index] = bottom;
}
```

Top-down reheapify (sink / trickleDown)

When a node's key is smaller than one or both of its children's key



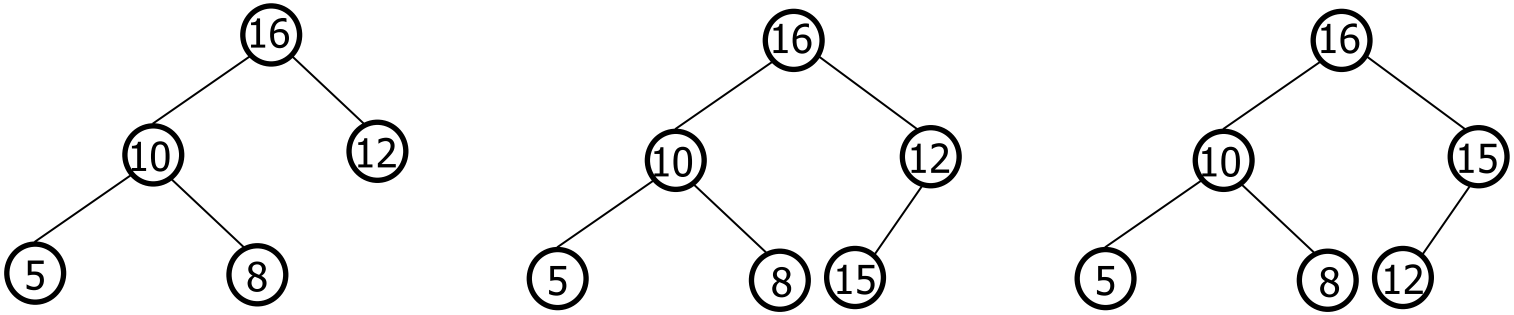
Top-down reheapify (sink / trickleDown)



```
public void trickleDown(int index)
   {
   int largerChild;
   Node top = heapArray[index];       // save root
   while(index < currentSize/2)       // while node has at
      {                               //    least one child,
      int leftChild = 2*index+1;
      int rightChild = leftChild+1;
                                      // find larger child
      if(rightChild < currentSize &&  // (rightChild exists?)
                          heapArray[leftChild].getKey() <
                          heapArray[rightChild].getKey())
         largerChild = rightChild;
      else
         largerChild = leftChild;
                                      // top >= largerChild?
      if( top.getKey() >= heapArray[largerChild].getKey() )
         break;
                                      // shift child up
      heapArray[index] = heapArray[largerChild];
      index = largerChild;            // go down
      }  // end while
   heapArray[index] = top;            // root to index
   }  // end trickleDown()
```

Insertion in a Heap

- insert(15) in the following heap:



- Where to put the new element?
  - At the end, or the first open slot in the array.
  - But, to keep the heap property, move the new element up until it is in the correct place.

Insertion in a Heap

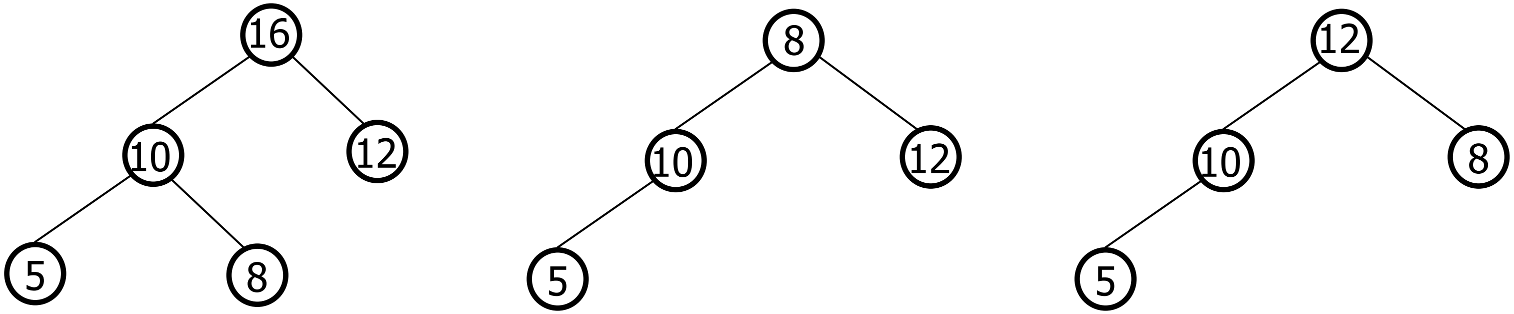
```
public boolean insert(int key)
{
   if(currentSize==maxSize)
      return false;
   Node newNode = new Node(key);
   heapArray[currentSize] = newNode;
   trickleUp(currentSize++);
   return true;
}
private void trickleUp(int index)
{
   int parent = (index-1) / 2;
   Node bottom = heapArray[index];

   while( index > 0 && heapArray[parent].getKey() <
bottom.getKey() )
   {
     heapArray[index] = heapArray[parent];
     index = parent;
     parent = (parent-1) / 2;
   }

   heapArray[index] = bottom;
}
```

What is the time complexity?

O(log n)

Removal from a Heap

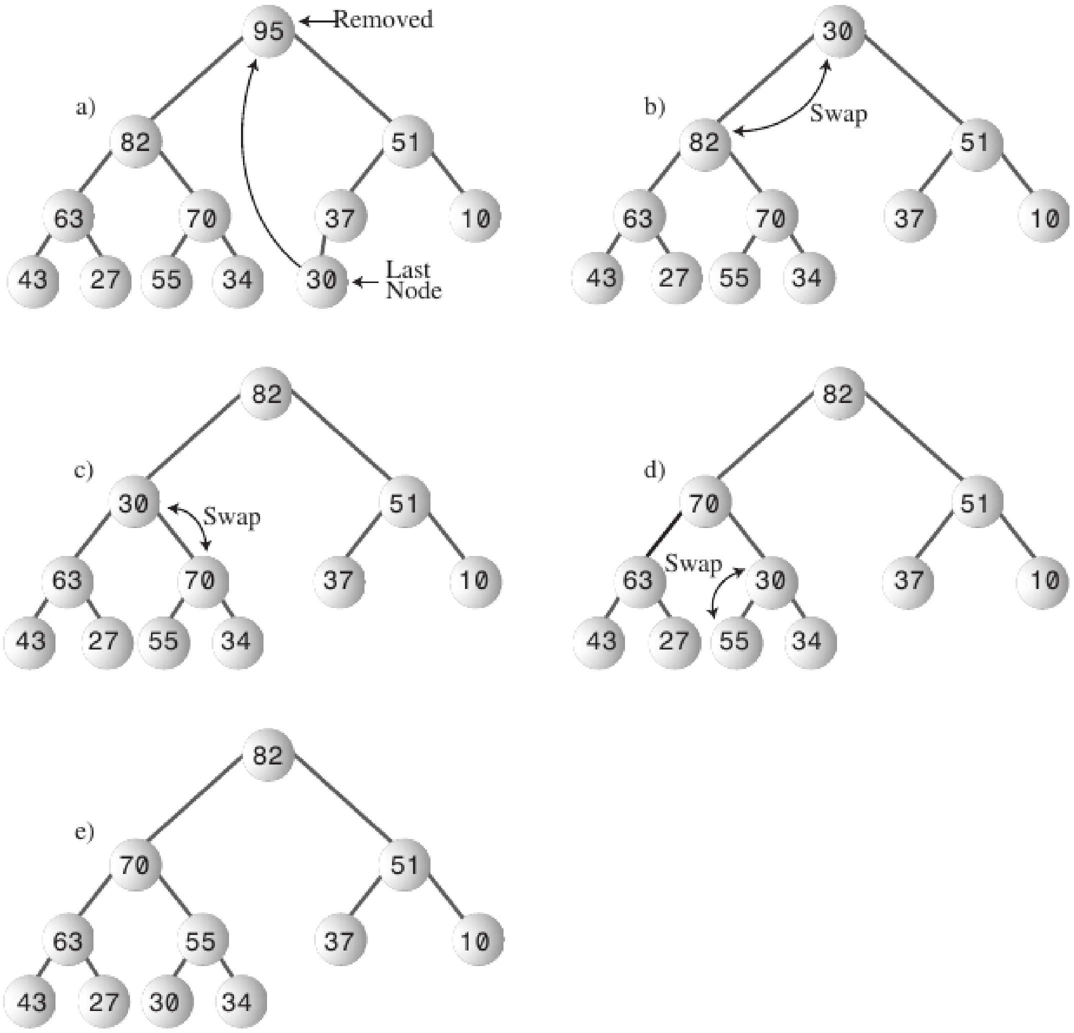
- Remove() in the following heap:



Which child to swap with?

- We remove the root.
- What should we put in its place?
  - The last node.
  - But, to keep the heap property, move the new element down until it is in the correct place.

Removal from a Heap-example



How many swaps?

Depends on the height. O(log n)

Robert Lafore. 2002. Data Structures and Algorithms in Java (2 ed.). Sams, Indianapolis, IN, USA. Page 584

Removal from a Heap

```
public Node remove()
{
   Node root = heapArray[0];
   heapArray[0] = heapArray[--
currentSize];
   trickleDown(0);
   return root;
}
```

Robert Lafore. 2002. Data Structures and Algorithms in Java (2 ed.). Sams, Indianapolis, IN, USA. Page 594

Removal from a Heap

```
public void trickleDown(int index)
{
   int largerChild;
   Node top = heapArray[index]; // save root
   while(index < currentSize/2) // while node has at
   { // least one child,
      int leftChild = 2*index+1;
      int rightChild = leftChild+1;
      // find larger child
      if(rightChild < currentSize && // (rightChild exists?)
          heapArray[leftChild].getKey() < heapArray[rightChild].getKey())
             largerChild = rightChild;
      else
             largerChild = leftChild;
      // top >= largerChild?
      if( top.getKey() >= heapArray[largerChild].getKey() )
          break;
      // shift child up
      heapArray[index] = heapArray[largerChild];
      index = largerChild; // go down
   } // end while
   heapArray[index] = top; // root to index
}
```

Robert Lafore. 2002. Data Structures and Algorithms in Java (2 ed.). Sams, Indianapolis, IN, USA. Page 594